Python for Data processing

Lecture 1:
Jupyter, Arrays, tensors and computations

Gleb Ivashkevich

whoami

Gleb Ivashkevich

doing **deep learning** - time series, satellite imagery

PhD in theoretical physics

6 years in **academia** doing numerical simulations

6 years in **data science** and **machine learning**

datarythmics **effimly**

data driven manufacturing efficiency

Why Python?

Python is:

- simple enough
- flexible
- general purpose
- has huge ecosystem for DS and ML

Why Python?

But it's interpreted! **Isn't it slow?**

Short answer is: **No. It's ok.**

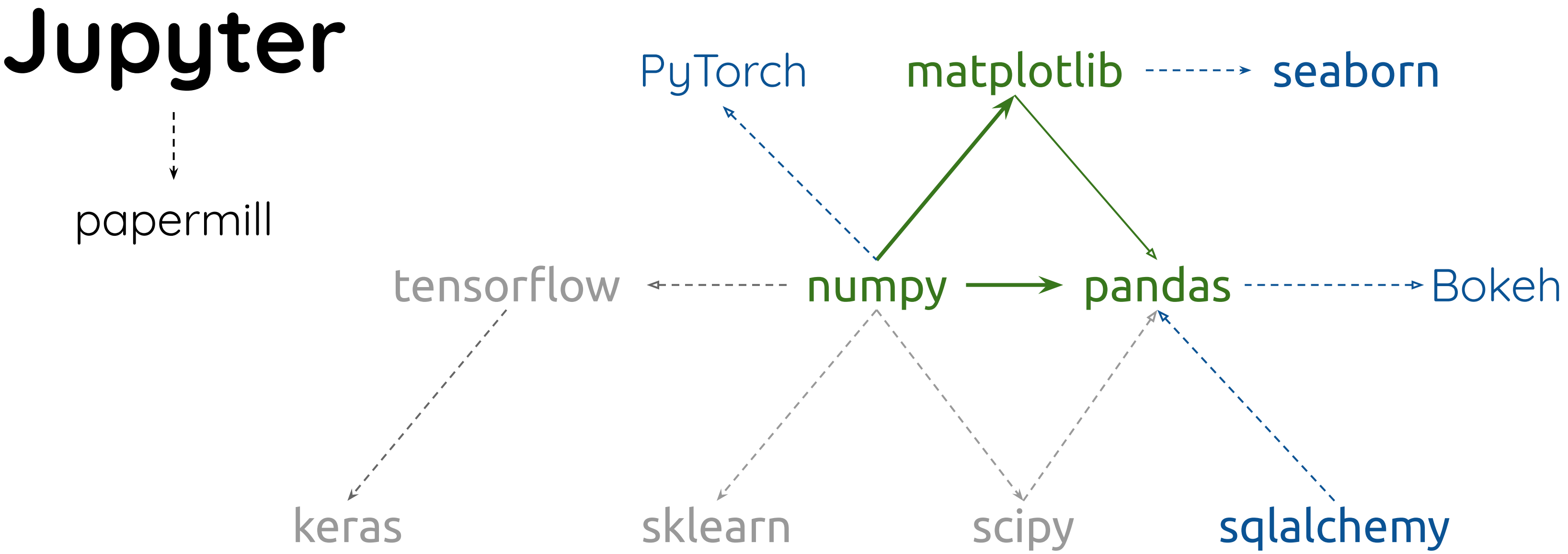
Long answer is: **No, it's not slow,** cause all the heavy lifting is done in **C/C++/Fortran** under the hood. Thank you, Python C API!

Syllabus

Main parts of the course are:

- **numpy** (PyTorch as a topping)
- **matplotlib** (+ `seaborn` and Bokeh)
- **pandas**
- basics of exploratory data analysis
- tools for reproducibility, project structuring and more

Python ecosystem for ML



Course logistics

Each week:

- lecture slides (released on Mon, topic based)
- Jupyter notebooks (released on Mon, topic based)
- one graded assignment (released on Wed, week based)
- one or more optional assignments (released during a week)

\+ online discussions

Course logistics

Graded assignments:

- we run them with papermill
- they should not fail
- partially autograded
- you have two weeks

Course logistics

Study groups:

- Nov 1: form study groups (if you don't, we assign you randomly)
- Nov 5: tell us, if randomly assigned group is not working for you (logistics, whatever)
- do homework together, discuss, have fun

→let's try it out!

(i.e. we're going to switch to notebook, terminal or whatever)

Resources worth reading

Python for Data Analysis by Wes McKinney

PyData YouTube channel (https://www.youtube.com/user/PyDataTV)

From Python to Numpy by Nicolas P. Rougier

http://www.labri.fr/perso/nrougier/from-python-to-numpy/

...and there will be more along the way.

Jupyter and other tools

Jupyter

web-based interactive environment

extremely suitable for exploration

originate in IPython project, but is largely **language agnostic now**

→let's try it out!

Jupyter: a bit of safety

Jupyter server, when running in the cloud/remote machine **should not be open** to the world

Use password and **https** or **ssh** tunneling

numpy:
basics of high performance arrays

Why numpy?

Pure Python:

- is slow (everything works through Python interpreter)
- lacks strong numerical infrastructure (**math** module? really?)

numpy:

- fast (it's C/C++/Fortran and battle tested BLAS etc. implementations)
- a lot of routines for virtually any generic use case

ndarray

- core data structure in **numpy**
- container with **known number** of elements of the **same** (known) **size**
- supports **indexing** and **vectorized** operations
- allows to **share** data

Creating arrays: naive

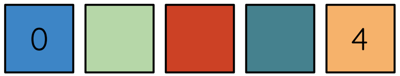
- let’s use **np.ndarray** directly!
- Or, better, let’s create an array from Python sequence

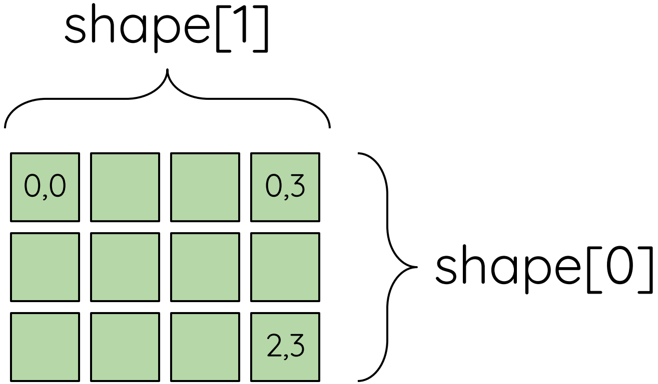
→let’s try it out!

Array: basic properties

- shape: `arr.shape`
- type of elements: `arr.dtype`, `arr.itemsize`
- number of dimensions: `arr.ndim`, `ar.size`

Python list vs. ndarray





len = 5
element size = ?

shape = (3, 4) (**elements**)
ndim = 2
element size = fixed

Creating arrays: advanced

We rarely need to create arrays from Python sequences (and **np.ndarray** should be avoided altogether)

Instead we need:

- arrays of **specific structure or type**
- arrays, filled with some **numeric pattern**

→let's try it out!

Array: basic indexing

numpy arrays support slicing syntax:

- **a[0, 1]** is ok
- **a[0, :3]** is ok
- **a[1:, :3]** is ok
- **a[0, :-9]** and even this is also ok

→let's try it out!

Array: boolean and fancy indexing

Basic indexing may be (and for large arrays it usually is) insufficient:

- boolean: **a[boolean_mask]**
- fancy: **a[int_idx]** (remember about **np.where**)

→let’s try it out!

Array: view vs. copy

Basic indexing returns **view,** fancy and boolean indexing return new **new array**.

But you can mix them.

And it's a bit different for setting values in an array.

→let's try it out!

Array: changing shape

Sometimes we need to change array shape:

- flat vector to row or column vector
- row vector to column vector
- transpose

`arr.reshape, np.expand_dims, arr.T`

`arr.flatten, arr.ravel`

Array: changing type

Sometimes we need to **change array type**:

- to create integer mask
- to reduce memory consumption
- to conform with external API

Array: stack

Sometimes we need to combine multiple arrays:

- to create a matrix from several vectors
- to combine results from different sources

→let's try it out!

Array: universal functions

Fast, vectorized functions, operating element-wise.

- **unary: np.sum, np.mean** and so on
- **binary: np.maximum, np.logical_and** and so on

Array: universal functions

ufuncs support common arguments:

- **axis**: operate over this axis
- **where**: masking
- **keepdims**: do not drop reduced dimensions

numpy from inside

ndarray from inside

numpy array

is a container:

```
typedef struct PyArrayObject {
    PyObject_HEAD
    char *data; /* Block of memory */
    PyArray_Descr *descr; /* Data type descriptor */
    /* Indexing scheme */
    int nd;
    npy_intp *dimensions;
    npy_intp *strides;
    /* Other stuff */
    PyObject *base;
    int flags;
    PyObject *weakreflist;

} PyArrayObject;
```

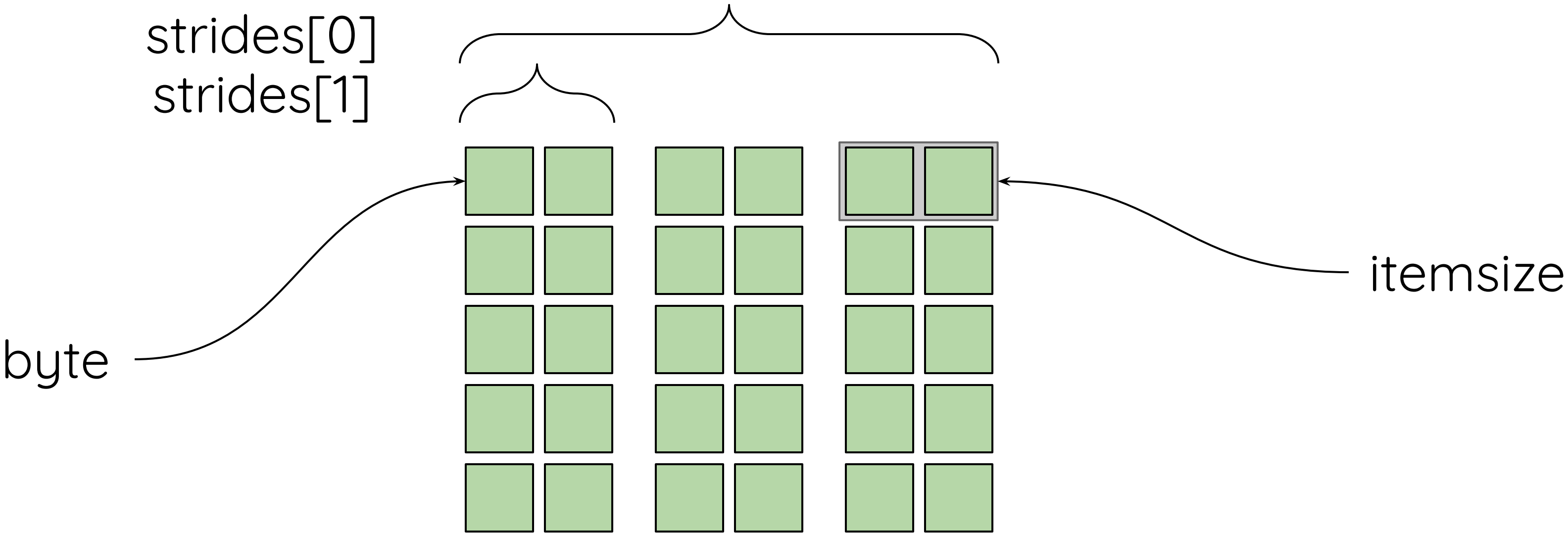
ndarray from inside

numpy:

- stores data as a flat chunk of memory
- have indexing scheme on top of that (dimensions and type)
- knows how to step through the memory
- knows the origin

→let's try it out!

ndarray from inside



→let's try it out!

shape = (5, 3) (**elements**)
strides = (6, 2) (**bytes**)
(i, j) = i * strides[0] + j*strides[1]

Consequence #1: cache effects

Data is read from memory in chunks, not element by element

↓

Memory layout may impact performance

→let's try it out!

Consequence #2: copies

Copies are costly

↓

Look at inplace operations

→let's try it out!

Efficient `numpy`

- use indexing wisely
- avoid loops
- use broadcasting whenever possible
- avoid copies whenever possible
- use inplace operations whenever possible
- **vectorize**

Still slow?

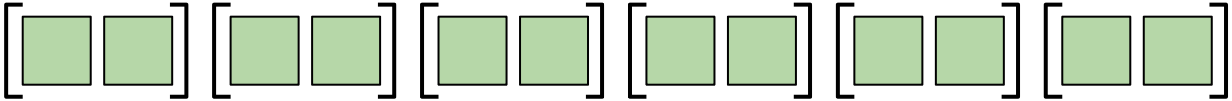
What if the code is so complex, it **gains little** from all the remedies above?

We have tools for that also.

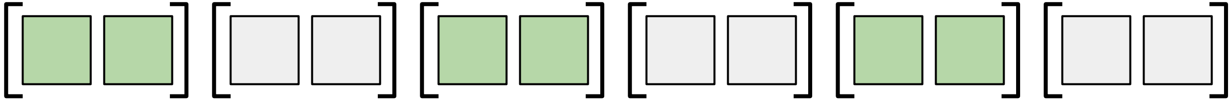
Cython, Numba → optional assignment

View vs. copy

`arr`$^{\texttt{np.int16}}$



`arr[::2]`



`arr`$^{\texttt{np.int16}}$

shape = (6,) **strides** = (2,)

(i) = i * 2

`arr[::2]`

shape = (3,) **strides** = (4,)

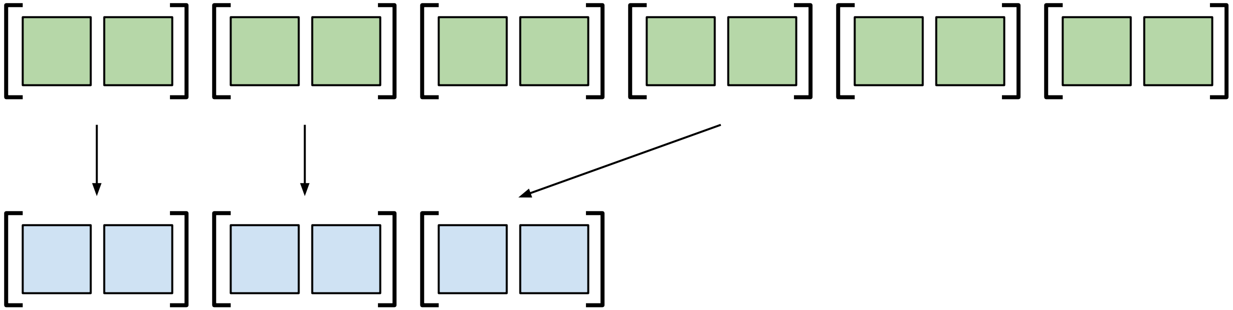
(i) = i * 4

`arr[::2]` is a **view**

View vs. copy

`arr`$^{\texttt{np.int16}}$

`arr[[T, T, F, T, F, F]]`



`arr`$^{\texttt{np.int16}}$

shape = (6,) **strides** = (2,)
(i) = i * 2

`arr[::2]`

shape = (3,) **strides** = (2,)
(i) = i * 2

`arr[::2]`
is a **copy**

Broadcasting

What if input arrays have **different shapes**?

- should we reshape them to common shape before applying some **ufunc**? **No.**
- if possible, ufunc adds missing dimensions and loop through them with stride=0

→let's try it out!

numpy:
basics of linear algebra

Linear algebra: basics

Linear algebra works on vectors (1D), matrices (2D) and tensors (>3D).

Typical operations:

- dot-product of vector and matrix
- matrix operations: invert, get eigenvalues
- decompositions

Linear algebra: np.linalg

Entry point for linear algebra operations:

- np.linalg.inv, np.linalg.det, np.linalg.trace
- np.linalg.eig
- matrix decompositions

→let's try it out!

Linear algebra: eigenvalues and eigenvectors

The simplest possible decomposition for square matrices

Plays huge role in many algorithms (although, in extended ways)

→let's try it out!

Reading and writing data with **numpy**

Reading and writing **numpy** array

Array can be saved to a file with **np.save**:

- binary format
- read with **np.load**

→let’s try it out!

Reading and writing numpy arrays

Multiple array can be saved to a single file with **np.savez**:

- it's zip, but uncompressed
- use **np.load** to read it (return dict-like object, no data is actually read)

→let's try it out!

Reading and writing text files

np.loadtxt and **np.savetxt** are used to read text files (mostly CSV)

But **pandas** is much better in this!

→let's try it out!

Other formats and options

Natively or through scipy.io:

- binary data (from files)
- **mat** files
- wav files

Using 3rd party packages:

- HDF5 (**h5py**)
- images (**skimage**, **opencv**)

What we’ve learned

- creating and indexing arrays
- changing array properties
- calculate with arrays
- basics of linear algebra operations
- I/O

Assignment

- exploring `numpy`: array creation, indexing
- broadcasting
- `ufuncs`

questions?